A Hard Real-Time Capable Multi-Core SMT Processor

Marco Paolieri, Barcelona Supercomputing Center (BSC) and Universitat Politècnica de Catalunya
Jörg Mische, University of Augsburg
Stefan Metzlaff, University of Augsburg
Mike Gerdes, University of Augsburg
Eduardo Quiñones, Barcelona Supercomputing Center
Sascha Uhrig, Technical University of Dortmund
Theo Ungerer, University of Augsburg
Francisco J. Cazorla, BSC and Spanish National Research Council (IIIA-CSIC)

Manuscript received 14 June 2011. Accepted for publication on 20 November 2011.

Hard real-time applications in safety critical domains require high performance and time analyzability. Multi-core processors are an answer to these demands, however task interferences make multi-cores more difficult to analyze from a worst-case execution time point of view than single-core processors. We propose a multi-core SMT processor that ensures a bounded maximum delay a task can suffer due to inter-task interferences. Multiple hard real-time tasks can be executed on different cores together with additional non real-time tasks. Our evaluation shows that the proposed MERASA multi-core provides predictability for hard real-time tasks and also high performance for non hard real-time tasks.

Categories and Subject Descriptors: C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms: Design, Performance

Additional Key Words and Phrases: Multi-core, SMT, Real-Time, Worst-Case Execution Time

ACM Reference Format:
Paolieri, M., Mische, J., Metzlaff, S., Gerdes, M., Quiñones, E., Uhrig, S., Ungerer, T., and Cazorla, F.J. 2011. A Hard Real-Time Capable Multi-Core SMT Processor. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January 2013), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Higher performance than what current embedded processors offer will allow hard real-time embedded systems to increase safety, comfort and the number and quality of services in many domains such as avionics, space, automotive, off-road vehicles, power plants, and automation [Wolf 2007; De Bosschere et al. 2007].

In the general-purpose computing market, high performance has been achieved in the past by designing complex processors with long pipelines, out of order execution, or with high clock frequency. However, in the design of hard real-time embedded systems these techniques are never feasible: On the one hand, complex processors with out of order execution pipelines suffer from timing anomalies [Lundqvist and Stenstrom 1999], which heavily complicates the computation of safe worst-case execution time

This work has been supported by the EC Grant Agreement n° 216415 (MERASA).

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

(WCET) estimations. On the other hand, the high energy requirements of complex processors running at high frequencies do not satisfy the low-power constraints and cost limitations of common embedded systems. Therefore relatively simple single-core processors are applied in real-time embedded systems, which are networked to up to 70 Electronic Control Units (ECUs) in a state-of-the-art passenger car.

Multi-cores provide high performance by placing multiple cores into a single chip. The potential benefits of multi-core processors for real-time embedded systems are multiple. First, providing higher performance than state-of-the-art processors will allow future embedded systems to provide more functionalities. Second, if simple cores are applied, these are easier to analyze. Third, multi-cores provide good performance/Watt ratio, which is mandatory in embedded systems. And fourth, multi-cores can maximize hardware utilization, reduce costs, size, weight, and energy consumption by co-hosting applications with different requirements. Several ECUs may be combined within one multi-core in the automotive domain, and – as demand of avionics and space domains – the execution of mixed-criticality application workloads comprised of hard real-time (HRT) and non hard real-time (NHRT) tasks will be enabled.

State-of-the-art multi-cores, however, have an important drawback which makes their use difficult or even impossible in hard real-time embedded systems: it is hard to perform WCET analysis due to inter-task interferences. Inter-task interferences appear when two or more tasks that share a given hardware resource try to access it at the same time. To prevent conflicts, an arbitration policy is required then. It selects which task gets the access granted to such shared resource, potentially increasing the execution time of other tasks and consequently their WCETs. Therefore, the WCET of a task depends on the inter-task interferences introduced by the co-running tasks. This affects the *timing composability*, which is a key property of integrated systems architectures, such as Integrated Modular Avionics (IMA) in the avionics domain. Timing composability means that the WCET behavior of individual tasks is not changed by the composition when the system is integrated. If the WCET of an HRT task were dependent on the co-running tasks, a change in any other task would require re-analyzing and re-certifying the system, incurring in high costs. Hence, a major design goal for a time analyzable multi-core architecture is to make the WCET estimations provided for each task independent from the other tasks in the task set it may be co-scheduled with, when the system is integrated.

In this paper we propose the MERASA architecture, a hard real-time capable multi-core architecture with simultaneously multithreading (SMT) cores. The objectives of the architecture are twofold: First, it must enable performing the WCET analysis using current measurement-based [Rapita Systems Ltd. 2008] and static [Cassé and Sainrat 2006] analysis tools. We accomplish this requirement by designing the architecture from a WCET point of view: The SMT core is enabled to simultaneously execute a single HRT task in concert with up to three NHRT tasks. It provides full isolation within a core such that the execution of the HRT task is never blocked by the NHRT tasks. Moreover, the MERASA multi-core processor provides time bounded interaction between HRT tasks running on different cores. Thus it achieves timing composability. A second major objective of our architecture is to provide good performance for the NHRT tasks exploiting all resources not used by HRT tasks, which is obtained by deploying SMT cores designed to fully isolate execution of the HRT task from the NHRT tasks with respect to time analyzability.

This paper on the MERASA multi-core describes all solutions applied within the SMT cores and at multi-core level to accomplish a hard real-time capable multi-core processor. It shows how the solutions are composed to achieve the design objectives of time analyzability and high-performance. The MERASA multi-core is the first and up to now only hard real-time capable multi-core processor available as FPGA hardware prototype, evaluated by simulations, modeled in a static WCET tool (OTAWA) [Cassé

and Sainrat 2006], enhanced to enable usage of a measurement-based WCET tool (RapiTime) [Rapita Systems Ltd. 2008], and evaluated by an industrial application, a collision avoidance algorithm provided by Honeywell International. A short overview on the MERASA processor architecture, its toolchain and results of a static WCET analysis has been published in [Ungerer et al. 2010]. In this paper, we target multiprogrammed workloads composed of mixed-criticality applications[1]. We do not cover the execution of parallel HRT application: this is the main goal of a follow-up project called *parMERASA*[2]. However, some preliminary works have been published in [Rochange et al. 2010], [Paolieri et al. 2011], and [Gerdes et al. 2011].

Core level: Each core [Mische et al. 2010] is designed as in-order SMT processor prioritizing one HRT over the up to three NHRT tasks. For time analyzability multiple changes over standard SMT cores were necessary. The main characteristics of the proposed core architecture are:

(1) An in-order superscalar pipeline as basis that combines good performance with the ease of analyzability of the HRT task.
(2) Complete isolation of the HRT task from the other tasks by granting the highest priority for every instruction of the HRT task in all affected pipeline stages (fetch, issue, memory arbitration).
(3) A data scratchpad and a dynamic hardware-managed instruction scratchpad set aside for the HRT task to ease WCET analysis.
(4) First-level caches for the NHRT tasks for high performance.
(5) Blocking instructions are avoided by a split-phase memory access and the implementation of long-running operations as interruptible microcodes.

Multi-core level: HRT tasks running on different cores have to access common resources, in particular the memory, in a time-bounded style [Paolieri et al. 2009]. To reach this goal the following contributions are proposed by this paper:

(1) A Real-Time Bus, in which the maximum delay an HRT task can suffer due to the requests of other tasks is bounded.
(2) A multibank Dynamically Partitioned Cache where the data and instructions of the different HRT tasks are put apart to prevent any interaction.
(3) An HRT capable memory controller for dynamic RAMs, which is shared among all HRT and NHRT tasks. The HRT capable memory controller cannot suffer from unpredictable refresh cycles that would lead to overestimation of the WCET.
(4) A WCET Computation Mode that allows the use of measurement-based WCET analysis for a multi-core processor.

It is important to notice that the solution proposed for on-chip and off-chip resources can be applied to I/O resources [Gerdes et al. 2011].

This paper is structured as follows: In Sections 2 and 3 we describe in detail the MERASA core and multi-core solutions to provide time analyzability. In Section 4 and 5 we present the experimental environment and the results on evaluation of average case performance, WCET and logic utilization of an implementation of the MERASA multi-core. Section 6 summarizes the related work and finally Section 7 shows the main conclusions of this paper.

2. CORE DESIGN

2.1. Core Design Overview

The main challenge at core level is to eliminate inter-task interferences that may hinder time analyzability of the HRT task. In fact, a WCET analysis of the HRT task

[1]Please note that each application is composed of a single task.
[2]www.parmerasa.eu

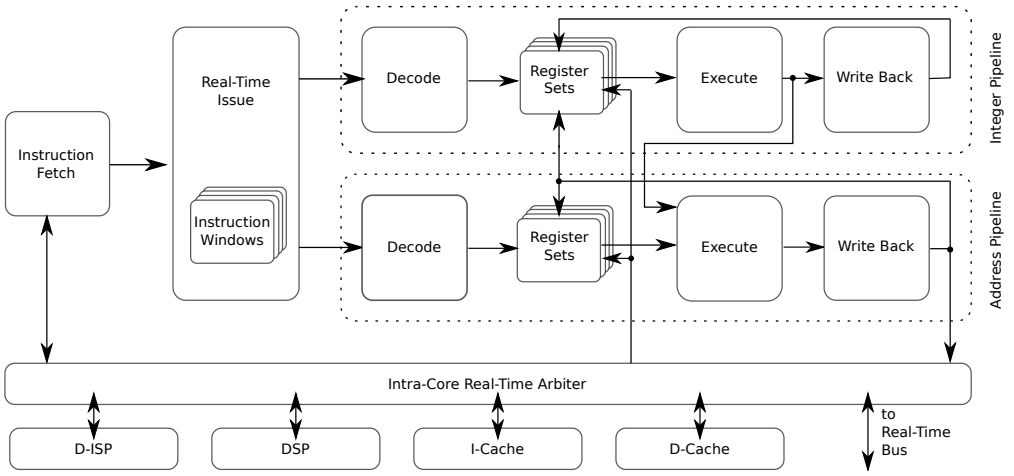


Fig. 1. SMT core architecture with a 5-stage dual-issue pipeline

should be possible as if it were the only task running on the SMT core. Our design goal is to minimize the WCET of the HRT capable highest priority task, while releasing as many resources as possible for the execution of concurrent non critical tasks.

We decided to provide four thread slots[3] (separate instruction pointers, instruction windows and register sets per thread slot) within the SMT core and allow the simultaneous execution of one HRT task and up to three NHRT tasks. Evaluations showed that the application of more thread slots only marginally improves the throughput [Mische et al. 2008]. We further chose to base our SMT design on an in-order superscalar processor pipeline, because out-of-order pipelines complicate the design, hinder WCET analysis and regarding SMT processors, the performance of in-order and out-of-order pipelines is comparable [Hily and Seznec 1998].

The basic pipeline design (see Figure 1) resembles the TriCore processor of Infineon Technologies, which is an in-order superscalar single-core processor (without multithreading) that is widely applied in hard real-time automotive domains. Each core implements two pipelines, an integer and an address pipeline. Both pipelines consist of five stages, where the first two (Instruction Fetch and Real-Time Issue) are shared while the last three (Decode, Execute and Write Back) are independent for each pipeline. The instructions are issued in-order and two instructions from one task can be issued in parallel, if an address instruction directly follows an integer instruction, otherwise the pipelines are filled by instructions from different tasks. The MERASA core implements the Infineon TriCore's instruction set [Infineon Technologies AG 2008] which allows us to use the TriCore toolchain.

2.2. Instruction Fetch and Real-Time Issue Stages

The tasks in the thread slots are scheduled by a fixed priority preemptive (FPP) algorithm: each task has a fixed priority and the task with the highest priority that is ready is executed. If there is a HRT task running on a core, it has the highest priority while other NHRT tasks have lower priorities. The prioritization policy privileges the HRT task in the Instruction Fetch stage, the Real-Time Issue stage and the Intra-Core Real-Time Arbiter (see Figure 1).

[3]The term thread is more commonly used in the multithreaded processor domain and task in the real-time domain. The thread slots of an SMT core may be filled by threads or tasks, however, we only consider independent tasks in this paper.

The Instruction Fetch stage is designed to guarantee that lower priority tasks do not delay the HRT capable highest priority task. Fetch bandwidth and Instruction Window size are chosen such that optimal execution of the highest priority task can be guaranteed. The highest priority task only requires the full fetch bandwidth, if its code occupies every pipeline in every cycle and if these instructions are of maximum length. Evaluations in [Mische et al. 2010] showed that this is almost never the case. Whenever the Instruction Window of a thread slot is full, the fetch logic tries to fetch for the task with the next highest priority.

The Real-Time Issue (RTI) stage receives the fetched instructions from the fetch stage and inserts them into the Instruction Window of the appropriate thread slot. Then the RTI analyzes and assigns the instructions to the two pipelines. The RTI issues instructions from the Instruction Windows by their priority. If the next instruction of the HRT task cannot be issued, an instruction of a lower priority task is issued to the pipelines.

Additionally the RTI manages multi-cycle instructions. In order to enable the isolation, multi-cycle instructions are implemented as interruptible microcode sequences. This interruptibility is important, otherwise low priority tasks would delay higher priority tasks for several cycles, once they were able to start their microcode sequence.

2.3. Handling Memory Operations

If a memory access takes multiple cycles, single threaded processors stall the complete pipeline until the access is completed. Such a behavior in a multithreaded processor would prevent all other tasks from being executed, even the HRT task with the highest priority. Hence, the Intra-Core Real-Time Arbiter (ICRTA) (see Figure 1) that serves as memory arbiter has to issue memory accesses in the same prioritized order like the RTI issues instructions. Additionally it must guarantee the complete isolation of the HRT task.

We applied a technique called Split Phase Load. A load instruction is split into two instructions, the address calculation and the register write back. When the RTI stage recognizes a load instruction, it issues the first part of the instruction that calculates the address in the Execute stage and forwards it to the ICRTA, which is responsible for the access to the Real-Time Bus (see Section 3.1). Later when the ICRTA receives the data, it notifies the RTI stage to issue the second phase of the instruction that writes the data from the memory to the register set in the Write Back stage. When writing to memory, the data word is transferred in conjunction with the address. As nothing must be written back to the register set, the second phase can be omitted for store instructions.

After issuing a memory operation to the first level memory, either a store or the first phase of a load, the task is temporarily suspended from issuing further instructions. When the memory instruction arrives at the ICRTA the address is saved in the *address buffer* of the appropriate thread slot (there is one address buffer per thread slot available). Whenever a memory operation is completed, the ICRTA scans the address buffers in priority order and starts a memory operation, if there is a valid entry. At the same cycle the ICRTA notifies the RTI stage to resume issuing instructions of the task whose data word had just arrived. Depending on the kind of instruction the RTI stage continues with the second phase of a load instruction or the next instruction after a store.

There is another advantage of the Split Phase Load / Address Buffer technique: it is possible to handle variable memory latencies. If the memory access is fast, the second phase of the load is issued earlier, if it takes longer, the write back instruction (respectively the next instruction after a store) is issued later. So memory accesses that are delayed by other cores can be handled inherently.

2.4. On-core Local Memories

To prevent inter-task interferences that disrupt the analyzability of HRT tasks, accesses to data and instruction memory of HRT tasks are separated from NHRT memory accesses. To that end, the architecture features first level data and instruction caches for the NHRT tasks and first level scratchpads for the HRT tasks (see Figure 1).

Two local scratchpad memories are dedicated to the HRT task in order to reduce the WCET for fetches and data accesses, because any memory access that is handled locally does not use the bus and will not suffer inter-task interferences. A Dynamic Instruction Scratchpad (D–ISP) [Metzlaff et al. 2008] loads the complete code of an activated function dynamically on call and return. The dynamic content management is necessary, since code often does not fully fit in the core local memories – especially for multi-core processors with restricted on-chip memory sizes. In this case the D–ISP can provide higher performance than static and software-controlled scratchpads. Evaluations show that the D–ISP performance is close to the performance of an instruction cache, however the D–ISP provides a better time predictability [Metzlaff et al. 2011].

Furthermore, a Data Scratchpad (DSP) is used for the stack which is frequently accessed and can be maintained locally. All the other data (like the heap data) is stored in the higher memory levels (either in the Dynamically Partitioned Cache or – in case of a miss – in the DDRx SDRAM memory device). By design the contents of the D–ISP and the DSP are completely independent of the behavior of the NHRT tasks.

2.5. Dynamic Instruction Scratchpad

The D–ISP stores complete functions of the HRT task in a fast core-local scratchpad memory. Therefore the D–ISP bundles all instructions of a function as one entity in the scratchpad. Analogous to a cache line in a cache, a function can only be read or replaced as a whole. The D–ISP loads instructions on demand and evicts parts of its content if the free space in the scratchpad memory is not enough. This is in contrast to conventional instruction scratchpads, which either use a static assignment or dynamically load the code by software routines.

The penalty by dynamically loading the function is kept small and predictable. Moreover to ease WCET analysis it is ensured that data and fetch accesses cannot interfere with each other within one core. The reason is that the operation of the D–ISP is two phased: during execution of a function all fetch requests are handled by the D–ISP and no data access can be delayed by any fetch request. During the function load phase the D–ISP stalls the processor and therefore no data accesses can occur. Thus the time analysis for fetches and data memory accesses can be done separately. If we could not exclude such conflict, a higher worst-case memory access time for fetches and data accesses caused by the interference has to be taken into account. This would lead to a quite pessimistic WCET estimation, since not all data memory accesses will conflict with instruction memory accesses and vice versa.

The structural overview of the D–ISP is shown in Figure 2. It consists of a D–ISP controller, the function memory, a mapping table, a lookup table, and a content stack. The D–ISP controller contains a fetch control, the content management, and the context register.

The fetch control handles the fetch requests and delivers the requested instructions to the pipeline. Therefore the fetch control needs to be aware of the currently active function and at which address in the function memory it is stored. The context register is used to look up the currently active function address. It contains the address of the active function in the native address space, its address in the function memory, and the function size.

The main task of the content management is to ensure that the necessary instructions for the incoming fetch requests can be handled by the fetch control. Therefore the content management checks on function activation (via call or return) the content

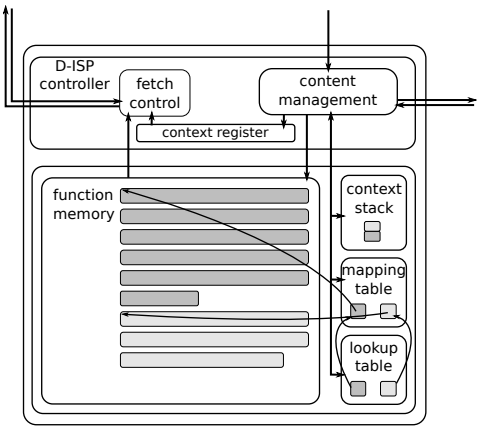


Fig. 2. D–ISP

of the function memory and stores the mapping information for the function into the context register. The content management distinguishes two cases: either the function is already present in the function memory or not. In case that the function is in the function memory, the content management obtains the address and size information from the mapping table which holds the native addresses, the function memory addresses and the sizes of all functions that are currently stored in the function memory. The content management writes these information for the activated function into the context register such that the fetch control is able to start serving fetch requests for that function.

On function call, the function address is directly given by the target of the call instruction (calculated by the ALU in the address pipeline). Using this address the content management starts the lookup for an entry in the mapping table. On return, the D–ISP controller has to determine the function address of the caller function itself. This is not as simple as for calls, because the address of the return point is not the address of the function (that is stored in the mapping table). To allow the determination of the function address on return the D–ISP content management maintains a function address stack as helper memory – the so-called context stack which is depicted in Figure 2.

To speedup function address lookup a special lookup-table assigns the mapping table position to the function address. Its associative structure allows to lookup the entries in parallel. If a function is found in the lookup table, the corresponding entry in the mapping table can be accessed directly.

If a function is not found in the mapping table on its activation, the content management requests the needed blocks from the ICRTA. The ICRTA uses the Real-Time Bus to obtain the requested blocks from the higher memory levels (either the Dynamically Partitioned Cache or the DDRx SDRAM memory device). The received blocks are copied by the content manager into the function memory. After all blocks are copied to the function memory the context register is written and the fetch control triggered to start function execution.

To load a function completely into the function memory the D–ISP content management has to know its size. For determining the function's size we chose to instrument the functions. We created an instrumentation tool, that hooks into the compilation and linking process of the application. The instrumentation tool adds a special instruction at the beginning of every function in the application code, which encodes the function length. Using this instruction the content management obtains the function size and copies the appropriate number of bytes into the function memory. Since the D–ISP

stores only complete functions it has to be at least as large as the largest function that is intended to be loaded into the function memory. Functions that are not instrumented or that are larger than the function memory will not be loaded into the D–ISP.

If the D–ISP content management loads a function that is larger than the unused function memory space, it has to decide which part of the function memory has to be overwritten. We propose the FIFO replacement policy for the D–ISP, since it is implementable with a low hardware effort and also analyzable, although it is known that it is not the replacement policy with the lowest miss rate. The implementation of the FIFO policy in the content management is done by cyclic addressing of the function memory space. Every fetch word is written to the address in the function memory that is selected by a write-pointer, which is increased after write and reseted, if the function memory address space is left. On overwriting the first fetch word of a function in the function memory the corresponding mapping table and lookup table entries are deleted, thereby the whole function is evicted. The address of the function that is evicted next is easy to determine, because the address of the next write into the function memory is known in advance.

For optimal usage of the D–ISP the programmer must be provided with Programming Guidelines that state the size of the function memory of the D–ISP (all time critical functions should be small enough to fit into the function memory) and the size of the mapping table, which restricts the number of functions concurrently available in the function memory. We suggest a mapping table size of 128 functions, which is large for an embedded application. Of course applications with more concurrently active functions are supported, causing function eviction on table overrun. If the number of associative comparisons in the lookup-table is restricted to 32 instead of 128, the function lookup can be performed within 4 cycles with a reasonable hardware effort [Metzlaff et al. 2011].

3. MULTI-CORE PROCESSOR DESIGN

In this section we focus on the design of the MERASA multi-core architecture, that is shown in Figure 3. Each core is connected through a *Real-Time Bus* to a *Dynamically Partitioned Cache* [Paolieri et al. 2009]. Such shared cache is interfaced through a Real-Time Capable Memory Controller (RTCMC) to a DDRx memory device. Those hardware shared resources are the source of inter-task interferences, and hence they have been designed being real-time capable.

3.1. The Real-Time Bus

An on-chip bus is commonly characterized by a *latency*, that is the amount of time required by the data to travel across the bus. When a request gets the access granted to the bus, no other request can use it. An arbitration policy is required when two or more requests try to access the bus simultaneously. In such scenario, one task will delay the others (i.e. the bus is busy and so it cannot be used by the other tasks) generating a *bus interference*.

In Figure 4 we show an overall picture of our Real-Time Bus, it is designed such that it is possible to take into account the worst-case scenario in terms of bus interferences during the WCET analysis. Our Real-Time Bus implements a two-level arbitration scheme, composed of: Inter-Core Bus Arbiter (XCBA) that schedules requests from different cores, and several Intra-Core Real-Time Arbiters (ICRTAs), responsible for scheduling requests within each core (see section 2.2 for further details). The key factor of our design is: it ensures that the delay a task can suffer due to the requests of any other task is upper bounded.

In the MERASA architecture, cores send a request to the bus on (1) D–ISP fills, (2) any load/store outside the DSP for HRT tasks, (3) every data cache load miss, (4) instruction cache miss and (5) store operation for NHRT tasks. These requests are

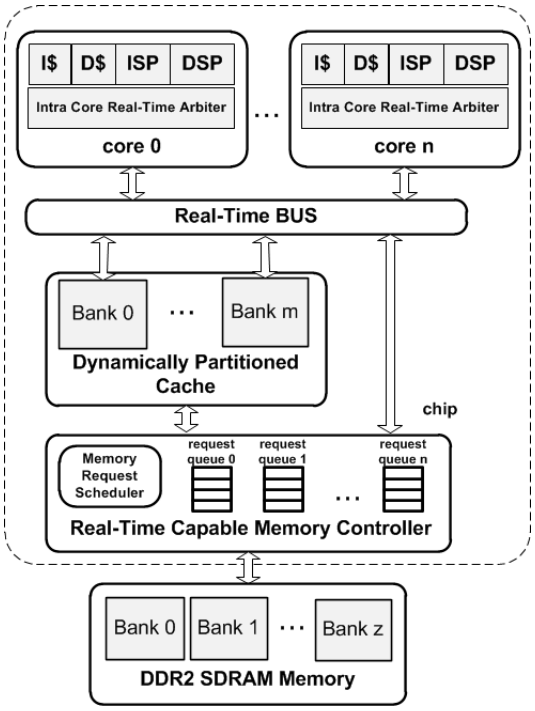


Fig. 3. General multi-core MERASA architecture

handled by the core's corresponding ICRTA, which selects the next memory request to be sent to the XCBA. The XCBA selects which core with a pending request can access the bus. The requests from the different cores are stored in separated queues and the XCBA considers only the top of such queues. Hence, the execution time, and so the WCET of a task depends only on the number of cores and not on the total number of requests from the other tasks that are ready and waiting to get the access to the bus (as it would happen in case of a global queue shared among the cores).

In our architecture, and along this paper, we assume, without loss of generality, that XCBA needs 1 cycle to select which request accesses the bus (labeled as A in following figures), 2 additional cycles are necessary for the data to cross it (labeled as B in figures and L_{bus} in equations) and 4 cycles to access a bank of the Dynamically Partitioned Cache (an access to bank n is labeled M_n in figures and L_{bank} in equations).

Although not shown in Figure 4, our Real-Time Bus is full-duplex and so the same approach is applied to the arbiter that controls the requests that are sent back from the Dynamically Partitioned Cache to the cores.

In order to satisfy our second objective of achieving high performance, each ICRTA has different *bank request queues* where the requests to the Dynamically Partitioned Cache are stored. Each private bank request queue stores the requests based on their target destination bank. Each queue contains the information about each memory request and the index to the data buffer entry that stores all the data corresponding to such request. When a core sends a request, the ICRTA time-stamps it and inserts it into its corresponding bank request queue. The ICRTA implements an arbitration policy to select the next memory request that is going to be forwarded to the XCBA. In particular, the ICRTA applies the following policies among requests from different bank queues:

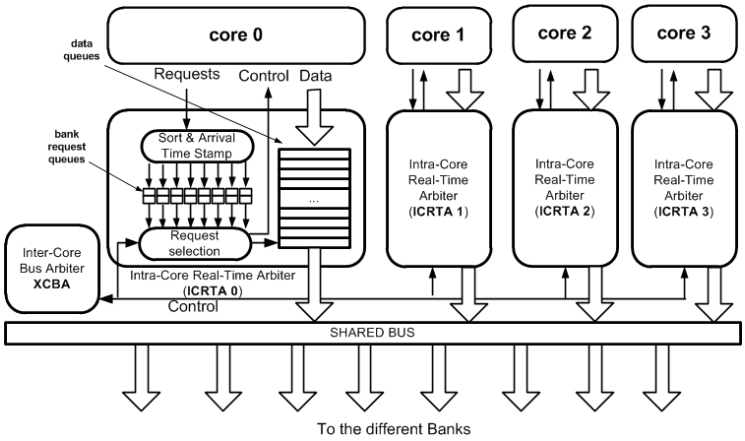


Fig. 4. The Real-Time Bus

In the case of HRT tasks, to prevent timing anomalies [Lundqvist and Stenstrom 1999] that can be originated by an out of order execution, the requests are sent in order by applying a FIFO policy. In case of NHRT tasks, the parallel out of order execution of different cache requests that do not address the same bank is allowed, in order to increase the overall performance. Hence a *First Ready First Serviced* policy is used. Bus transfers wider than the bus bandwidth are split into several independent requests that can be sent in non-consecutive bus slots.

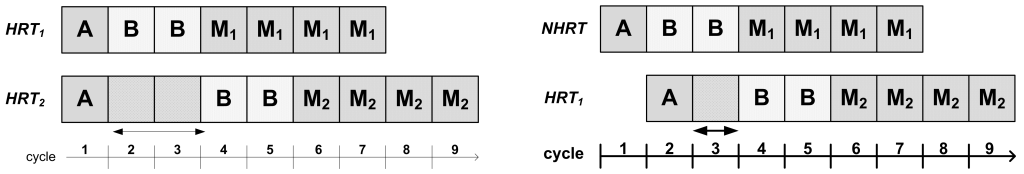
Since the delay a task can suffer due to bus interferences depends on the bus arbitration policy, in [Paolieri et al. 2009] we analyze the effect that different arbitration policies applied to the Real-Time Bus may have on the WCET of an HRT task. The XCBA enforces that a request from an HRT task cannot be delayed longer than a given Upper Bound Delay (UBD), and we define formally such UBD. Throughout this section we assume that each task has its own private memory controller, so different tasks do not interact accessing the memory; hence tasks are only affected by bus interferences. In Section 3.2 we complete our study considering also the effects of a shared cache.

When multiple HRT tasks are running simultaneously inside the processor, it may happen that two or more requests from different HRT tasks try to access the bus at the same time. By applying a *round robin* arbitration policy, the maximum delay a request from an HRT task can suffer is bounded by the total number of HRT tasks that can send a request concurrently. Figure 5(a) shows an example of such worst-case scenario that occurs when two different HRT tasks issue two requests at the same time. In this case, an HRT task HRT_2 must wait until the previous request from HRT_1 finishes. The maximum delay HRT_2 suffers, is: $UBD_{BUS}^{HRT} = (N_{HRT} - 1) \cdot L_{bus}$, where N_{HRT} is the number of co-running HRT tasks which is upper bounded by the number of cores.

In addition to that, it may happen that the request from the HRT task arrives just one cycle after one request from a NHRT task (i.e. it got already granted the access to the bus). In such scenario, the request from the HRT task will be delayed by the request from the NHRT task (see Figure 5(b)). The maximum delay that a request from the HRT task can suffer due to a NHRT request is upper bounded by the following expression: $UBD_{BUS}^{NHRT} = L_{bus} - 1$

Our multi-core processor implements a round robin bus arbitration policy between HRT tasks and prioritizes them over NHRT tasks. The maximum delay is determined by the combination of the effects of the requests coming from HRT tasks and NHRT tasks:

$$UBD_{BUS} = UBD_{BUS}^{HRT} + UBD_{BUS}^{NHRT} = L_{bus} - 1 + (N_{HRT} - 1) \cdot L_{bus} \quad (1)$$



(a) Worst-case scenario between two HRT tasks

(b) Worst-case scenario between NHRT & HRT task

Fig. 5. Example of interference accessing the bus

That can be simplified as follows:

$$UBD_{BUS} = N_{HRT} \cdot L_{bus} - 1 \tag{2}$$

We want to highlight that the N_{HRT} is the number of HRT tasks running at the same time inside the processor (which is upper bounded by the number of cores) and not the total number of HRT tasks that compose the system.

3.2. The Dynamically Partitioned Cache

In this section, we add to our analysis the study of the interferences caused by a shared cache. The use of shared memories in multi-core systems introduces unpredictable and not analyzable worst-case behavior due two factors: *bank access interference* and *storage interference*.

Bank Access Interference: Caches are normally partitioned into multiple banks in order to enable parallel accesses: different memory operations can access different banks simultaneously. However, a bank can only handle one memory request at a time. When a bank is serving a memory request, this bank is inaccessible to any other request for an amount of cycles equal to the *bank latency*. So, if two memory requests try to access the same bank simultaneously, the XCBA avoids any conflict by delaying the second access. This kind of effect, called *bank interference* or *bank conflict*, may introduce variability in the execution time of a task.

Storage Interferences: *Storage interferences* appear in shared caches when one task evicts useful data of another one, causing a potential delay the execution time of the second task (with respect to execution in isolation) due to additional misses originated. Such behaviour makes the WCET analysis harder or even infeasible.

Bankization: Our solution is the use of *cache partitioning*, that is a well known technique that can eliminate completely storage interferences by splitting the cache into private partitions, each of them assigned to a different task. Our mixed workload environment can benefit from cache partitioning: Storage interferences between HRT tasks are avoided by assigning them different partitions of the cache, while NHRT tasks can share the same part of the cache.

To exploit parallelism between different memory operations, shared caches are physically partitioned into banks. We take advantage of this design choice, to implement our Bankization partitioning technique. With Bankization, each task is assigned a subset of the set of banks that no other task can use. By using a simple bit vector, we can specify the set of banks assigned to a given task. Whenever a cache access is performed the bit vector is used to determine which is the target bank of the access. A given range of bits of the memory address is used to select the destination bank. Since Bankization assigns a subset of the cache banks to a given task, it is necessary to remap the destination bank of a memory request to one of the banks assigned to the task.

The additional hardware necessary to implement Bankization into our Dynamically Partitioned Cache, is based on a Bank Remapping Unit (BRU) that computes the target cache bank given the task identifier and the memory address. The BRU simply

changes the bits used to select the destination bank. In particular, it is composed of a small table having few entries (as many as cores). The assignment of tasks to banks can be changed dynamically at run-time by modifying the content of such table. The table inside the BRU is updated by the RTOS based on the subset of banks assigned to each task. The remapping table is indexed by the task id and the original bank id of a memory request. The BRU output is the new bank id.

With Bankization we prevent both storage and bank access conflicts, so the UBD can be computed with Equation (2).

3.3. The Real-Time Capable Memory Controller

A DDRx SDRAM memory system [Jacob et al. 2008] is composed of a *memory controller* and one or more *memory devices*. The *DRAM memory controller* is the component that controls, by using different commands, the off-chip memory system acting as the interface between the processor and memory device, while the memory devices store the data.

Our Real-Time Capable Memory Controller (RTCMC) design [Paolieri et al. 2009] is the result of an exhaustive analysis of the UBD introduced by memory interferences, considering the generic timing constraints defined in the JEDEC standard [JEDEC Solid State Technology Association 2008]. The RTCMC implements a *round robin* policy among HRT tasks so inter-task interferences are upper bounded based on the maximum number of tasks that can access simultaneously the memory (i.e. the total number of cores). It also prioritizes HRT tasks over NHRT tasks so the effect of NHRT tasks on the WCET estimation is reduced. Moreover, in order to isolate intra-task interferences (originated by requests of the same task) from inter-task interferences (originated by requests of different tasks), our memory controller uses a private request queue per task. By doing this, the RTCMC prevents any interaction between the requests of different tasks. Therefore, the maximum delay that a request can suffer because of other requests depends only on the number of queues (cores), i.e. $N_{\text{HRT}} - 1$.

The RTCMC uses an interleaved-bank address mapping scheme: every memory request access is split into an access to every bank (this paper considers a 4-bank DRAM device), so DRAM commands can be effectively pipelined. In addition to that, the impact of interferences is further reduced by using a *close-page* policy with *auto-precharge*.

The requests that the different cores issue to the memory are characterized by the *Request Inter-Task Delay* and the *Request Execution Time*. The former represents the delay the memory request can suffer — due to interferences with other requests generated by co-running tasks executing on different cores — before getting the access granted to the memory device. The latter identifies the time it takes for the request to be executed, once it is ready and it cannot suffer interferences with the other tasks. We defined an analytical model to compute an Upper Bound Delay for the memory (UBD_{MEM}) that bounds the *Request Inter-Task Delay*. Taking such UBD into account during the WCET analysis of an HRT, the WCET estimations will be independent from the rest of the co-running tasks. Timing composability is hence ensured.

As shown in [Paolieri et al. 2009], UBD_{MEM} is defined by the following Equation (3), in which $t_{ILWORST}$ is the worst-case *Request Inter-Task Delay*.

$$UBD_{MEM} = UBD_{MEM-NHRT} + UBD_{MEM-HRT} = N_{HRT} \cdot t_{ILWORST} - 1 \qquad (3)$$

3.4. WCET Computation Mode

In order to consider the UBD of each shared resource, UBD_{MEM} for the memory and UBD_{BUS} for the shared bus, during the measurement-based WCET analysis we extend our processor with a hardware feature called *WCET Computation Mode* [Paolieri et al. 2009]. In this execution mode, each HRT task is run in isolation: On each access to a shared resource (both the on-chip shared bus and the DRAM memory controller),

the processor artificially delays that access by the maximum time that a request from an HRT task can suffer because of inter-task interference, UBD_{BUS} for the bus and UBD_{MEM} for an access to memory. The *WCET Computation Mode* only requires knowing the number of HRT tasks that run at the same time on the processor. As a consequence, the resulting WCET from the profile of this execution delivers an upper bound of the execution of the HRT task when it runs in *Standard Execution Mode* together with other tasks sharing processor resources. This satisfies the timing composability for the task set. Once a WCET estimation has been obtained for each HRT task, the processor is set back to *Standard Execution Mode*, in which no artificial delay is introduced.

When running in *Standard Execution mode* instructions that access shared resources may be executed before their estimated WCET, since they are not always suffering the worst inter-task interference scenario. In [Andrei et al. 2008; Rosen et al. 2007] it has been formally proven that executing an instruction before its estimated WCET is safe. As a consequence, the WCET estimation provided by RapiTime, is a valid upper bound for the execution time of the HRT tasks when they run in the multi-core processor sharing resources with other tasks.

The UBD artificially introduced by our *WCET Computation Mode* is computed using Equation (2) and (3) respectively for the shared bus and for the main memory. The UBD depends on the total number of HRT tasks running simultaneously in the processor, i.e. upper bounded by the number of cores. Thus, a different UBD value is used by the *WCET Computation Mode* depending on the number of HRT tasks the analyzed task is going to be co-scheduled with; hence, resulting in different WCET estimations. An HRT task that is co-scheduled with $N-1$ HRT tasks, is analyzed using a *WCET Computation Mode* of N, which results in a WCET estimation $WCET_N$.

Our *WCET Computation Mode* allows analyzing each HRT task in isolation, i.e. independently from the particular task set in which that task is going to be scheduled. For every HRT task we build a WCET-matrix. The WCET-matrix has as many entries as the product of the number of WCET Computation Modes by the number of cache partitions a task can be assigned. In our baseline, we have a total of 64 configurations, 4 WCET Computation Modes times 16 cache configurations (recall that in our case we can assign from 1 to 16 banks to each HRT task). This WCET-matrix can be computed in *isolation* for each HRT task. This process can be easily automated, in fact we do so to run the experiments of this paper. The WCET-matrix can be used by the allocation algorithm to select the best assignment of resources for the tasks in the tasks set as we have shown in [Paolieri et al. 2011].

The WCET Computation Mode can be used to perform WCET analysis using measurement-based approaches, while to support a static WCET analysis it is necessary to model the shared resources, e.g. by enforcing each request to suffer a delay equal to UBD cycles.

4. EXPERIMENTAL SETUP

4.1. MERASA Architecture Simulator

All experiments presented in this paper were carried out using an in-house cycle-accurate, execution-driven simulator running TriCore ISA binaries with support for the measurement-based WCET tool RapiTime.

The simulator models the MERASA architecture composed of up to four SMT cores. Each core implements an in-order dual-issue pipeline, and can be configured to provide up to four threads slots, allowing to execute one HRT task and up to three NHRT tasks. Concretely, each core has the following characteristics: 5-staged pipeline, no branch prediction, a fetch bandwidth of 8 byte per cycle with up to 4 instructions. The size of the instruction window is 24 bytes (containing 3-12 instructions) per thread slot. The

total number of registers is 32 for each thread slot: 16 registers for the data and 16 for the address pipeline.

In order to prevent uncontrolled inter-task interferences, each core implements a D–ISP and DSP for the HRT tasks and a first level data and instruction cache for the NHRT tasks. The size of the D–ISP is 16KB, and all application code can be mapped into the scratchpad, so no eviction is necessary. To quantify the impact of the dynamic content management of the D–ISP, Section 5.1 evaluates the effect of the replacement policy by varying the size of the D–ISP. The DSP, which contains the stack, has a size of 16KB such that the stack can be mapped completely into the DSP. The size of the instruction and data cache is 16KB for each. They are configured as a 4-way, 1-bank, 8-byte per line cache with 1 cycle access, write-through write-not-allocate policy and LRU replacement policy.

Each core, has access to the Dynamically Partitioned Cache through the full duplex Real-Time Bus. The Dynamically Partitioned Cache is organized in banks in order to avoid storage interferences, implements the *bankization* partition technique with a size of 128KB, 16-way, 16-banks, 32-byte per line, 4 cycles access, write-back write-allocate policy, LRU replacement policy. The total number of cycles for L2 hit is 9. A private cache partition is assigned to each HRT task and it contains the heap. NHRT tasks share the same cache partition which acts a second level cache.

Our simulation framework also models in detail our RTCMC interfaced to a DDR2-400B JEDEC-compliant 256Mb x16 DDR2 SDRAM devices composed of a single DIMM, single rank and a single 4-banks memory device. We assume a CPU frequency of 800MHz, being a CPU-SDRAM clock ratio of 4. The DRAM memory system is modeled using DRAMsim [Wang et al. 2005], which we integrated inside our simulation framework. DRAMsim is a well known C-based memory system simulator, highly-configurable and parameterizable that implements detailed timing models for different type of existing DRAM memory systems.

Finally, in order to consider the UBD of both, the Real-Time Bus and the RTCMC, on the WCET estimation, the simulator implements the WCET Computation Mode, which allows to use WCET analysis tools already used in single-core systems in multi-core systems without *any* change. As a matter of example, we used RapiTime for providing WCET estimation of the HRT tasks. In case of static analysis tools, it is only required to model a single core with the timing model of the WCET Computation Mode for shared resource access, instead of considering the whole multi-core system including inter-core interferences.

4.2. Benchmarks

Benchmarks were selected to evaluate mixed-criticality application workloads, i.e. applications with and without real-time constraints running on the same system. For the evaluation we used several benchmarks that are representative in both domains. To yield Infineon TriCore binaries the benchmarks were compiled with the commercial Altium TASKING compiler [Tasking 2005]. All benchmarks are independent tasks that can be executed concurrently.

Hard real-time applications: As HRT tasks we use benchmarks from the EEMBC AutoBench benchmark suite and small selection of applications from the Mälardalen and MiBench benchmark suites for the evaluation of the core memory hierarchy. In order to be representative of future HRT applications we increased the EEMBC memory requirements (enlarging the input data set, and without modifying the source code of the benchmarks). Moreover we have classified EEMBC benchmarks in Table I into three different groups according to their accesses to shared resources, i.e. according to their memory requirements: memory-intensive demanding group (labeled as *High*), medium memory-intensive demanding group (*Medium*) and a non memory-intensive demanding group (*Low*).

Table I. EEMBC Benchmark Classification by Memory Requirements

Memory Demand	Benchmarks
High	aifftr01, aiifft01,cacheb01
Medium	aifirf01, iirt01, matrix01, pntrch01
Low	a2time01, basefp01, bitmnp01, canrdr01, idctrn01, puwmod01, rspeed01, tblook01, ttsprk01

In addition to the benchmarks, we use an industrial HRT application provided by Honeywell International: The collision avoidance application is based on an algorithm for 3D path planning used in autonomous-driven vehicles and airplanes to process the frames captured by on-board cameras and to build the path to reach the target points while avoiding any obstacles. It requires high-performance with high-data rate throughput and underlies HRT constraints.

Finally, in order to study the impact of memory interferences on WCET estimations, we designed a memory-intensive synthetic benchmark called `opponent` in which each instruction is a store that systematically misses in the Dynamically Partitioned Cache and so it always accesses the DRAM memory system. Thus, HRT tasks that are co-scheduled with such task will suffer the maximum impact on their execution time. However, we cannot ensure that the opponent represents the worst-case scenario, because it does not represent the worst-case memory interference scenario that an HRT task could suffer. In fact, building an application that represents the actual worst-case opponent is hard as it depends on every particular HRT task under study, requiring a high effort to be built and verified.

Non real-time applications: As NHRT tasks we use benchmarks from MediaBench II, SPEC2006 CPU and MiBench Automotive benchmark suites which competes against HRT tasks for processor's resources. From MediaBench II we use `mpeg2dec`, from SPEC CPU2006 `bzip2`, from MiBench Automotive `qsort` and `susan`. Based on these benchmarks we composed two different NHRT workloads with 2 tasks: a memory-intensive pair `mpeg2dec - qsort`, which frequently misses in the Dynamically Partitioned Cache and so generating inter-task interferences in the memory system; and a CPU-intensive pair `susan-bzip2`.

5. EVALUATION RESULTS

From the core design point of view, we evaluate the reduction of off-core memory requests for the HRT task when using both D–ISP and DSP, and we expose the data that supports some of the design decisions in our design space exploration of MERASA architecture. From the processor point of view, we study the impact that inter-task interferences have on WCET estimation of HRT tasks, as well as the impact that HRT tasks have over NHRT tasks.

5.1. Reducing Off-Chip Memory Usage with Scratchpads

The DSP and D–ISP allow to effectively reduce the number of off-core requests, leading to a lower execution time and a reduction of contention on the Real-Time Bus. It also reduces the number of off-chip memory-accesses reducing the contention in the off-chip memory bandwidth.

Figure 6 depicts the normalized memory bandwidth needed by the set of benchmarks from the EEMBC AutoBench (`rspeed`, `ttsprk` and the average over all AuoBench applications labeled as *Avg.*), the Mälardalen benchmark suite (`ADPCM`, `Compress`) and MiBench (*Dijkstra*). Each benchmark is executed as HRT task, considering three different memory configurations: (1) DSP only, so instructions are fetched directly from SDRAM memory; (2) D–ISP only, so the stack is contained in SDRAM memory; and (3) using both D–ISP and DSP. The data shows that using the DSP only the memory bottleneck is reduced by around 10% for most benchmarks. In case of `EDN`, the usage of the DSP reduces the memory bottleneck up to 30%. When adding an D–ISP, with the

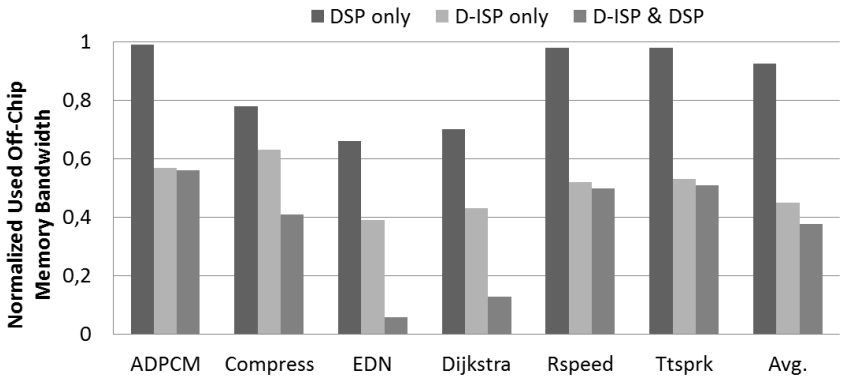


Fig. 6. Normalized used off-chip memory bandwidth with DSP, D–ISP and DSP & D–ISP with respect to not having any type of local memory (scratchpad).

size of the code such that no eviction takes place, to the DSP the shown benchmarks can reduce their needed memory bandwidth by at least 50%. For the average over all EEMBC benchmarks 60% bandwidth reduction is reached. Reducing the needed off-chip memory accesses by using the local scratchpads lowers the WCET as inter-task interferences are avoided, and fewer time consuming bus accesses are necessary.

Whereas the Figure 6 shows the maximum possible reduction of the used off-chip memory bandwidth by the D–ISP and DSP the Figure 7 depicts the penalty which is introduced if the D–ISP has a size smaller than the code size and thus eviction takes place. The figure shows the fetch cost in cycles needed when executing the benchmark, which are normalized to the case that every instruction is contained in a scratchpad before benchmark execution. So the normalized fetch cost of 1 cannot be reached since the code has to be copied into the D–ISP while benchmark execution, which needs additional fetch cycles. Because the CPU-SDRAM clock ratio is 4 a normalized fetch cost of 4 represents the case that the whole benchmark is executed from the SDRAM memory. To compare the behavior of the different benchmarks, we normalized the D–ISP size to the size of the benchmark, such that for the normalized size of 1 the whole benchmark fits into the D–ISP. In this case the depicted overhead due to the D–ISP is caused by loading the functions once into the D–ISP.

Since the D–ISP size has to be at least as large as the largest function in the benchmark the leftmost data for the different benchmarks presented in Figure 7 marks the size of the largest function in the code. For this configuration thrashing can be observed which even could lead to the fact that the D–ISP performs worse than if no on-chip memory is used, like for `ttsprk` (with a normalized fetch cost of 27.9).

The reason for the thrashing is the frequent reloading of functions and the fact that always the complete function has to be loaded – no matter if only a small part of it will be executed. It occurs if a function is nearly equally sized as the D–ISP, causing the eviction of every other function that is maintained by the D–ISP on its load. Also smaller functions will evict the largest function on their execution. To suppress the costly reloading of the largest function, the scratchpad size has to be increased. Increasing the D–ISP size beyond the size of the largest function leads to a close to optimal fetch cost that is shown in Figure 7.

The specific ratio of D–ISP size and code size where the D–ISP performs best depends on the code structure of the application. The EEMBC Benchmarks (`rspeed` and `ttsprk`) have a characteristic gradient of the fetch cost: the lowest D–ISP size leads to thrashing, whereas the next most larger size leaves to nearly optimal fetch cost. This is caused by the structure of the EEMBC application code: one function is very

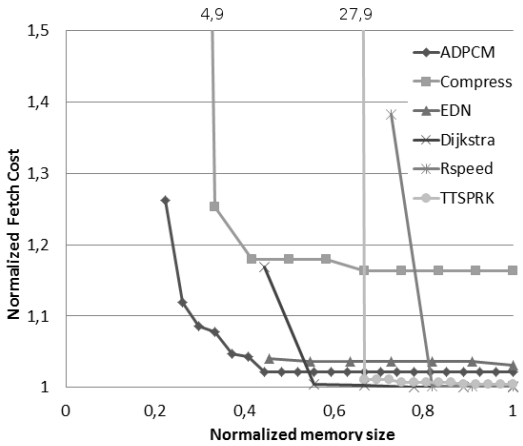


Fig. 7. Normalized fetch cost with varying D–ISP sizes. Normalized to the case that the whole code in in a scratchpad before its execution and the size of the application.

large (with respect to the code size) and the benchmark has a flat call hierarchy. To also inspect deeper call hierarchies and more balanced function sizes, we used a small subset of the Mälardalen and the MiBench benchmark suites. As shown for `ADPCM` or `Compress` the benefit of the D–ISP increases when the D–ISP size gets larger and more functions can be maintained by the scratchpad at once. The fetch cost saturates if either all functions can be hold in the D–ISP or active patterns in the call hierarchy (like a set of functions called within a loop) are executed without eviction.

The DSP complements the D–ISP that is based on the granularity of functions, since the DSP holds the stack data used by the parameters, return values and local variables of functions. Therefore the core architecture encourages the usage of functions in the application code for reducing the WCET of the HRT application. For an efficient usage of the D–ISP the functions in the code should be preferably equally sized, because functions nearly as large as the scratchpad size may cause thrashing and reduce overall (worst-case) performance of the application.

5.2. Multi-Core Processor Design Point Decision

As part of our research we performed a design space exploration to determine the architectural parameters, in particular the maximum number of NHRT tasks running on each core.

Figure 8 depicts the normalized IPC throughput of the MERASA multi-core processor configuring it as a single-, dual-, and quad-core processor with one to five thread slots running various EEMBC AutoBench benchmarks. The results show an increment in the overall throughput for all three processor versions when considering up to four thread slots. However, when increasing the number of thread slots to five slots, the overall throughput is reduced significantly. This reduction is considerably higher for the dual- and quad-core configuration than for the single-core version and results from the contention on the shared bus, concretely on the ICRTA and XCBA components (see Figure 4).

Therefore, the contention suffered due to shared resources between the different NHRT tasks in the thread slots limits the performance benefits obtained from the idle cycles introduced from other tasks. Hence, we have chosen our design supporting one HRT task and three NHRT tasks per core, i.e. offering four thread slots per core.

To quantify the maximum performance of the NHRT tasks that benefit from the resources not used by the highest priority HRT tasks, we analyzed their IPC through-

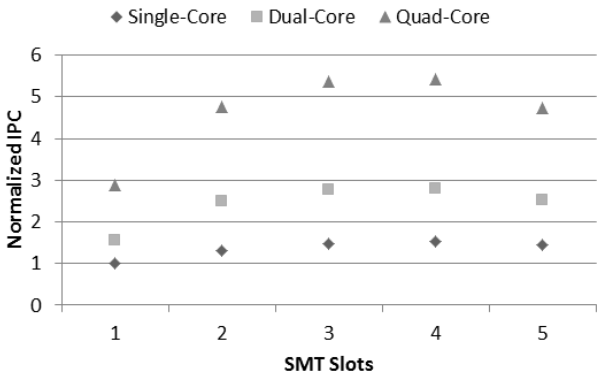


Fig. 8. Normalized IPC for different number of cores and thread slots configurations

put. We composed workloads with four tasks, two HRT tasks and two NHRT tasks running on a quad-core processor. As HRT tasks we use the two different memory demand groups: memory-intensive and CPU-intensive. We study the throughput of the NHRT tasks when they run simultaneously with other HRT tasks, normalized to the case when the NHRT tasks run with no other HRT tasks at the same time.

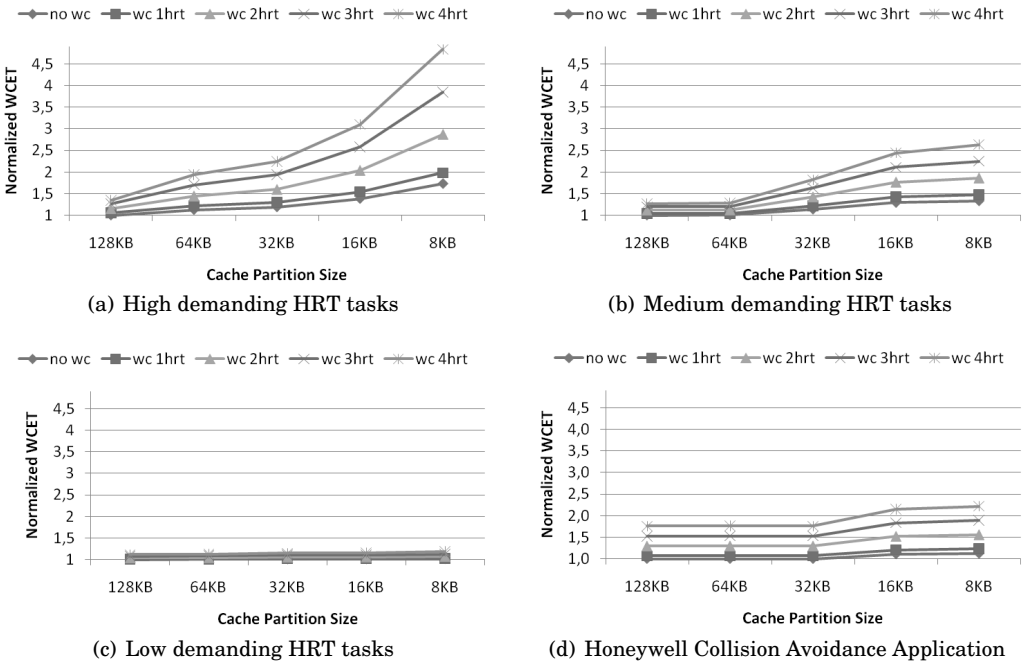
For memory-intensive NHRT tasks the normalized throughput ranges from 70% when they run with HRT tasks from Low group, up to 50% when they run with the memory-intensive HRT tasks from the High group. Instead, CPU-intensive NHRT tasks do not show any performance degradation. Their normalized throughput ranges between 97% and 99% as their number of misses in its corresponding cache partition is reduced. Hence, our architecture effectively allows NHRT tasks to exploit all the resources not used by the HRT tasks.

5.3. WCET Evaluation

This section analyzes the impact of inter-task interferences due to hardware shared resources on the WCET estimation. Concretely we study the WCET estimation increment of the three EEMBC benchmark groups (*High, Medium* and *Low*) and the Honeywell application when the pressure on the shared bus and the off-chip memory increases. That is, despite the benefits brought by the DSP and the D-ISP with a clear reduction of the inter-task interferences when accessing the shared bus and the off-chip memory, as the number of simultaneous HRT tasks is increased and the cache partition assigned to each HRT task is reduced, more inter-task interferences appear. All WCET estimation values have been computed using RapiTime with the WCET Computation Mode enabled. Values are normalized to the WCET estimation when the HRT task runs in isolation in the multi-core architecture and the WCET Computation Mode is disabled.

Figure 9 shows the average WCET estimation provided by RapiTime for *High*, *Medium* and *Low* memory demanding groups, and the Honeywell application as we vary the number of simultaneous HRT tasks from 1 to 4 and the amount of cache assigned to each of them for 128KB to 8KB. To do so, we use the WCET Computation Mode, in which a WCET Computation Mode N means that N HRT tasks are executing simultaneously on the processor (labeled as *wc 1hrt* for 1 HRT task , *wc 2hrt* for 2 HRT task, etc.) and bankization as the cache partition technique.

As expected, the *High* memory-demanding group, shown in Figure 9(a), is very sensitive to both the variation of the simultaneous HRT tasks and cache size reduction. On the one hand, when varying the WCET Computation Mode from 1 to 4 with 128KB



(a) High demanding HRT tasks

(b) Medium demanding HRT tasks

(c) Low demanding HRT tasks

(d) Honeywell Collision Avoidance Application

Fig. 9. Average WCET estimation of HRT tasks using a JEDEC DDR2-SDRAM 400B.

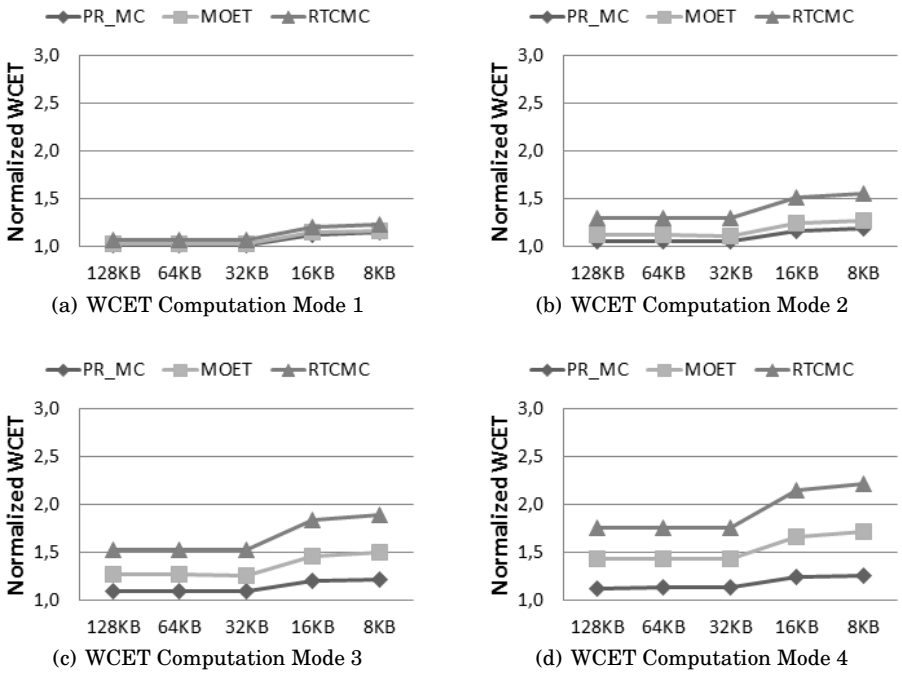
of cache, the WCET estimation increases with respect to not using it from 1.05x to 1.34x respectively. On the other hand, when reducing the cache size from 128KB to 8KB, and considering WCET Computation Mode of 4 the WCET estimation increases from 1.05x to 4.84x respectively. Hence, the inter-task interferences generated when running with a WCET Computation Mode of 4 and 8KB cache size, has a huge impact on the WCET estimation increases in comparison of not using WCET Computation Mode and having the whole cache.

The *Medium* memory-demanding group, shown in Figure 9(b), have a much more smooth behaviour in comparison to *high* demanding tasks. The WCET Computation Mode increases the WCET estimation from 1.04x to 1.26x when varying it from 1 to 4 with 128KB of cache. When running with WCET Computation Mode of 4 and 8KB cache size, the WCET estimation is increased to 2.63x in comparison of not using WCET Computation Mode and having the whole cache.

Finally, the *Low* memory-demanding group, shown in Figure 9(c), has a smaller impact due to inter-task interferences, increasing its WCET estimation, when running with WCET Computation Mode of 4 and 8KB cache size, by only 1.18x in comparison of not using WCET Computation Mode and having the whole cache.

The collision avoidance algorithm, provided by Honeywell and shown in Figure 9(d), resembles a *medium* demanding benchmark. A WCET Computation Mode of 4 and a cache partition of 8KB results in an overall increment of 2.22x in comparison to not using WCET Computation Mode and having the whole cache.

It is also interesting to analyze the impact of inter-task interferences generated by each shared resource on the WCET estimation. Figure 10 shows the WCET estimation of the Honeywell application under two different scenarios: (1) assuming a private DRAM memory controller for each HRT task and so having interferences only in the shared bus (labeled as *PR_MC*); and (2) bus and memory interferences are considered



(a) WCET Computation Mode 1

(b) WCET Computation Mode 2

(c) WCET Computation Mode 3

(d) WCET Computation Mode 4

Fig. 10. Normalized WCET estimation for the Hon application using JEDEC DDR2-400B

at the same time, using our interference-aware shared bus and the RTCMC shared among the different cores (labeled as *RTCMC*). Moreover, in order to evaluate whether the WCET estimations obtained using RTCMC are tight, we measure the Maximum Observed Execution Time (labeled as *MOET*) of the HRT task when running a very memory-intensive workload composed by several `opponents`.

For each scenario, we vary the WCET Computation Mode from 1 to 4, and for each WCET Computation Mode we vary the private cache partition size assigned to the Honeywell application from 128KB to 8KB. In case of the MOET, the number of `opponents` that compose the workload correspond to the number of the WCET Computation Mode.

As expected, memory interferences have a tremendous impact on the WCET estimation, significantly higher than bus interferences. In the scenario with the biggest amount of memory accesses, i.e. assigning 8KB of cache partition and a WCET Computation Mode of 4 (Figure 10(d)), the memory interferences increase the WCET estimations from 1.25 (PR_MC) to 2.22 (RTCMC), which represents a relative increment of 77%.

Even though memory interferences have such high impact, our architecture allows computing tight WCET estimations. When comparing the MOET and the WCET estimation (RTCMC) in the highest memory-intensive workload, i.e. assigning a cache partition of 8KB and running with 3 HRT `opponents` (Figure 10(c)), we observe an increment of 29% (from 1.72x to 2.22x). Note that, since the actual WCET of an HRT task is equal or bigger than any MOET and equal or smaller than any WCET estimation ($MOET \leq WCET_{actual} \leq WCET_{estimation}$) [Thiele and Wilhelm 2004], the WCET estimation obtained using our architecture is at most 29% bigger than the actual WCET.

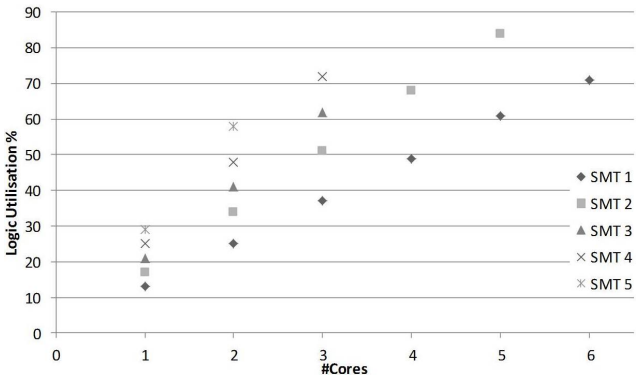


Fig. 11. Logic utilization of different core and thread slot configurations on a FPGA

5.4. Logic Utilization in an FPGA Implementation

To quantify the logic utilization of the MERASA multi-core design we explored the size of an implementation of our design on a Stratix II FPGA for a various number of cores and thread slots. The Figure 11 presents the logic utilization for some selected configurations with respect to the amount of cores and thread slots per core on a Stratix II (EP2S180F1020C3; 143,520 Adaptive Look-Up Tables (ALUTs) in total available) FPGA. In detail, Equation 4 shows the relation between the logic utilization and adding cores or thread slots.

$$logic\ utilization\ [in\ \%] \leq (6 + (5 \cdot slots)) \cdot cores \tag{4}$$

Figure 8 shows that adding a core to a given configuration results in a higher throughput than adding an thread slot. Naturally, adding an additional core also increases the logic utilization more than adding an thread slot (see Equation 4).

6. RELATED WORK

Egger et. al. [Egger et al. 2006] propose a managed instruction scratchpad, that combines static and software managed approaches: At compile time the functions are divided into three classes, depending on their location in the memory. Frequently used functions are placed in the scratchpad, rarely called ones are placed in the external memory. Some function in between are located in the external memory, but are loaded to the scratchpad on demand. The page manager, that is responsible for managing the dynamic part of the scratchpad is implemented in software and therefore much slower than our D-ISP solution.

The scratchpad manager described by Janapsatya et. al. [Janapsatya et al. 2006] is implemented in hardware and triggered by special copy instructions that are inserted into the code. The scratchpad contains basic blocks that are selected based on a temporal proximity metric. Basic blocks that are executed consecutively are assigned together to the scratchpad at the same time. In contrast to the D–ISP the scratchpad proposed by Janapsatya et. al. is not supposed to hide the memory hierarchy from the processors fetch path. Using the D–ISP every fetch is directed to it. This ensures that fetches does not interfere with other memory accesses, which is crucial for the WCET estimation.

The so-far mentioned approaches increase the analyzability by replacing the instruction cache by a scratchpad, but they were designed primarily to reduce energy consumptions. An instruction cache that was build for HRT systems with predictability in mind is the so-called method cache [Schoeberl 2004] of the Java processor JOP.

The D–ISP decouples the content and the function addresses for lookup into different structures, whereas the proposed cache structure binds a memory block to its cache tag. So a fine granularity of memory blocks to improve memory density, leads to a high number of cache tags that are causing either a slow or hardware intensive hit detection. In the D–ISP the scratchpad content is decoupled from the lookup tables. So the complexity of hit detection is restricted to the number of entries of the lookup tables only. Preußer et. al. address the complexity problem of a fine grained method cache implementation [Preußer et al. 2007] and show an more efficient implementation with a stack based replacement policy.

In order to execute multiple tasks in parallel supporting a predictable architecture, the concept of the Precision Timed (PRET) Machine [Lickly et al. 2008] was developed. In this approach, a precise timing and simplified WCET analysis is guaranteed by interleaved execution of six tasks. The architecture provides high throughput with a relatively slight hardware, but it is very inflexible: every task gets exactly one sixth of the execution time, it is neither possible to assign a larger share to a computation intensive task nor can the time, when a task completes earlier, be used for the other tasks.

The XCore architecure by XMOS Ltd. [May 2009] provides hard real-time capable multithreaded cores that can be combined by a hard real-time capable interconnect to a chip multiprocessor. But the cores are only single-issue and due to the interleaved multithreaded execution, a single task can use no more than 25% of the processor cycles. Within one core, memory is shared between the tasks, but inter-core communication is restricted to point-to-point messages.

The Real-time Virtual Multiprocessor (RVMP) architecture proposed by El-Haj-Mahmoud et al. [El-Haj-Mahmoud et al. 2005] virtualizes a single in-order superscalar processor into multiple interference-free different-sized virtual processors. The configuration of the virtual processors can be changed at run-time according to the timing requirements, providing a time analyzable architecture together with the flexibility of SMT processors. The processor partitioning is determined statically by the real-time scheduling framework that preserves the possibility of analyzing the WCET as in single-task processors. The architecture does not include any cache, that reduces the level of non-determinism, like in our approach.

In [Rosen et al. 2007] Rosen et al. described a solution to implement predictable real-time applications on multiprocessors. They propose a bus scheduling policy based on TDMA with a previously statically defined scheduling policy. Different time-slots to access the bus are allocated to different processor by static scheduling, i.e. stored in a memory directly connected to the bus arbiter. This technique needs to know the workload a priori, which is the whole set of tasks that run on the system at any given time, in order to avoid that the bus contention increases the memory access latency. This solution avoids sub-estimation of WCET due to bus conflicts and prevents any deadline miss. The architecture, which is used in [Khatib et al. 2006] for a real-time biomedical monitoring and analysis system, is a multi-core processor where each core has its own private memory, connecting all them with a bus.

All these [May 2009; El-Haj-Mahmoud et al. 2005; Rosen et al. 2007] approaches do not support any kind of memory hierarchy. Instead, they assume that everything fits in the scratchpad memory, which is not the case for high throughput data-rate applications, because their memory requirements are higher than commonly used scratchpad memories.

Another European project reconciling efficiency with predictability is called PREDATOR[4]. Its architectural approach faces analyzable caches, a compiler-controlled memory management and simple cores with analyzable behavior. Predictable kernel mech-

[4] www.predator-project.eu

anisms are used to cover the main challenge in the field of operating systems, i.e. the interference-caused variability in task execution times. The multi-core architecture is only a small part of this project. The aim of the project is to develop real-time analysis methods for an existing multi-core architecture and not the design of a new architecture with better real-time capabilities.

Caccamo et al. describe different aspects of accessing shared resources taking into account cache memories. In [Pellizzoni and Caccamo 2007] they deal with interferences at the bus level between cache accesses and I/O peripheral transactions, concluding that these kinds of inferences cause unpredictable behaviors. They present a theoretical framework able to model the interaction between the CPU and the peripherals accessing the front side bus. In [Bui et al. 2008] they address the cache partitioning problem (to avoid interference between different cores) as an optimization problem. The solution found by an optimization algorithm identifies the optimal size of each cache partition. Then the tasks are assigned to different partitions such that the worst-case utilization is minimized and the real-time schedulability is improved. However, contrary to our solution, tasks are required to know the characteristics of the workload in which they run, not accomplishing timing composability.

The CoMPSoC platform template [Hansson et al. 2009] is a template for composable and predictable multi-processor system on chip where each application is given its own reconfigurable virtual platform. Similar to our architecture, inter-task interferences among tasks are controlled in such a way that the timing behaviour of a task is independent of all other tasks, achieving timing composability. However, authors leave cache and off-chip SDRAM, considered in this paper, as part of future work.

Pitter and Schoeberl [Pitter and Schoeberl 2010] introduce a real-time Java multiprocessor called JopCMP. It is a symmetric shared-memory multiprocessor and consists of up to 8 Java Optimized Processor (JOP) cores, an arbitration control device, and a shared memory. All components are interconnected via a system on chip bus. An SRAM memory instead of a DDRx SDRAM memory is considered and each core is not SMT.

7. CONCLUSIONS

Multi-core processors can satisfy the high performance requirements demanded by embedded applications, with a consequent reduction in the number of processor chips, and so in the overall weight, cost and size of the system. However, inter-task interferences when accessing hardware shared resources prevent system designers from using contemporary multi-cores in safety-critical environments.

In this paper we have proposed hardware techniques that allow executing mixed-criticality multiprogrammed workloads in a SMT multi-core processor, providing safe WCET estimations for the HRT tasks and high performance for the NHRT tasks. The computation of safe WCET estimations for the time-critical tasks is enabled by full isolation of tasks and bounding of inter-task interferences. Moreover, the WCET analysis of each HRT task can be computed independently from the workload on the other cores.

Overall, we have shown that the design of a HRT capable multi-core processor with SMT cores is possible. We have proposed solutions at core level to enable SMT. In particular a highest priority HRT task is isolated from the other tasks running on the same core. Local scratchpad memories are dedicated to the HRT task in order to reduce the WCET for fetch operations and for data accesses, because any memory access that is handled locally reduces bus contention and inter-task interference. At multi-core level we have proposed an HRT capable bus and memory controller in which the maximum delay a task can suffer due to the requests of other tasks is bounded. We have used bankization as partitioning technique of the shared cache to prevent interaction between tasks.

Overall our results show that we provide predictability for HRT tasks running on an SMT multi-core processor with mixed-criticality workload, providing moderate WCET increase with respect to the Maximum Observed Execution Times. Our architecture provides high performance for NHRT tasks exploiting all resources not used by HRT tasks.

ACKNOWLEDGMENT

This work has been mainly funded by MERASA STREP-FP7 European Project under the grant agreement number 216415. Marco Paolieri is partially supported by the Catalan Ministry for Innovation, Universities and Enterprise of the Catalan Government and European Social Funds. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the grant Juan de la Cierva JCI2009-05455. Ministry of Science and Technology of Spain has been partially funding this work under contract TIN-2007-60625.

REFERENCES

ANDREI, A., ELES, P., PENG, Z., AND ROSEN, J. 2008. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*. IEEE Computer Society, Washington, DC, USA, 103–110.

BUI, B. D., CACCAMO, M., SHA, L., AND MARTINEZ, J. 2008. Impact of cache partitioning on multi-tasking real time embedded systems. *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08.*, 101–110.

CASSÉ, H. AND SAINRAT, P. 2006. OTAWA, a framework for experimenting WCET computations. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse*. SEE.

DE BOSSCHERE, K., LUK, W., MARTORELL, X., NAVARRO, N., O'BOYLE, M., PNEVMATIKATOS, D., RAMIREZ, A., SAINRAT, P., SEZNEC, A., STENSTROM, P., AND TEMAM, O. 2007. High-performance embedded architecture and compilation roadmap. 5–29.

EGGER, B., KIM, C., JANG, C., NAM, Y., LEE, J., AND MIN, S. L. 2006. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. CASES '06. ACM, New York, NY, USA, 223–233.

EL-HAJ-MAHMOUD, A., AL-ZAWAWI, A. S., ANANTARAMAN, A., AND ROTENBERG, E. 2005. Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 213–224.

GERDES, M., WOLF, J., GULIASHVILI, I., UNGERER, T., HOUSTON, M., BERNAT, G., SCHNITZLER, S., AND REGLER, H. 2011. Large Drilling Machine Control Code - Parallelisation and WCET Speedup. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 91 – 94.

HANSSON, A., GOOSSENS, K., BEKOOIJ, M., AND HUISKEN, J. 2009. CoMPSoC: A template for composable and predictable multi-processor system on chips. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. Vol. 14. ACM, 1–24.

HILY, S. AND SEZNEC, A. 1998. Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading. Tech. Rep. RR-3391, INRIA. .

Infineon Technologies AG 2008. *TriCore 1 User's Manual*. Infineon Technologies AG. V1.3.8.

JACOB, B., NG, S. W., AND WANG, D. T. 2008. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, San Francisco, CA, USA.

JANAPSATYA, A., IGNJATOVIĆ, A., AND PARAMESWARAN, S. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. IEEE Press, Piscataway, NJ, USA, 612–617.

JEDEC Solid State Technology Association 2008. *JEDEC DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*. JEDEC Solid State Technology Association.

KHATIB, I. A., POLETTI, F., BERTOZZI, D., BENINI, L., BECHARA, M., KHALIFEH, H., JANTSCH, A., AND NABIEV, R. 2006. A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: architectural design space exploration. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM, New York, NY, USA, 125–130.

LICKLY, B., LIU, I., KIM, S., PATEL, H. D., EDWARDS, S. A., AND LEE, E. A. 2008. Predictable programming on a precision timed architecture. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, New York, NY, USA, 137–146.

LUNDQVIST, T. AND STENSTROM, P. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium*. IEEE, 12–21.

MAY, D. 2009. *The XMOS XS1 Architecture* Ed. XMOS Ltd.

METZLAFF, S., GULIASHVILI, I., UHRIG, S., AND UNGERER, T. 2011. A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware. *Architecture of Computing Systems-ARCS 2011 6566*, 122–134.

METZLAFF, S., UHRIG, S., MISCHE, J., AND UNGERER, T. 2008. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *MEDEA '08: Proceedings of the 9th workshop on MEmory performance*. ACM, New York, NY, USA, 38–45.

MISCHE, J., GULIASHVILI, I., UHRIG, S., AND UNGERER, T. 2010. How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT. In *23rd International Conference on Architecture of Computing Systems (ARCS 2010), Hannover, Germany*.

MISCHE, J., UHRIG, S., KLUGE, F., AND UNGERER, T. 2008. Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads. In *IEEE International Conference on Computer Design 2008 (ICCD 08)*. Lake Tahoe, CA, USA, 371–376.

PAOLIERI, M., QUINONES, E., CAZORLA, F., BERNAT, G., AND VALERO, M. June 20-24, 2009. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *The 36th International Symposium on Computer Architecture (ISCA 2009). 2009, Austin, Texas*.

PAOLIERI, M., QUINONES, E., CAZORLA, F., DAVIS, R., AND VALERO, M. 2011. IA^3: an interference aware allocation algorithm for multicore hard real-time systems. In *Proceedings of RTAS*. 433–443.

PAOLIERI, M., QUINONES, E., CAZORLA, F., AND VALERO, M. 2009. An Analyzable Memory Controller for Hard Real-Time CMPs. *Embedded Systems Letters, IEEE 1,* 4, 86–90.

PAOLIERI, M., QUIONES, E., CAZORLA, F. J., WOLF, J., UNGERER, T., UHRIG, S., AND PETROV, Z. 2011. A software-pipelined approach to multicore execution of timing predictable multi-threaded hard real-time tasks. In *ISORC'11*. 233–240.

PELLIZZONI, R. AND CACCAMO, M. 2007. Toward the predictable integration of real-time cots based systems. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 73–82.

PITTER, C. AND SCHOEBERL, M. 2010. A real-time java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst. 10*, 9:1–9:34.

PREUSSER, T. B., ZABEL, M., AND SPALLEK, R. G. 2007. Bump-pointer method caching for embedded java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. JTRES '07. ACM, New York, NY, USA, 206–210.

RAPITA SYSTEMS LTD. 2008. RapiTime White Paper. `http://www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf`.

ROCHANGE, C., BONENFANT, A., SAINRAT, P., GERDES, M., WOLF, J., UNGERER, T., PETROV, Z., AND MIKULU, F. 2010. WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In *10th Int'l Workshop on Worst-Case Execution-Time Analysis in conjunction with the 22nd Euromicro Int'l Conference on Real-Time Systems, Brussels, Belgium*. Vol. 268. Austrian Computer Society, 92–102.

ROSEN, J., ANDREI, A., ELES, P., AND PENG, Z. 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. 28th IEEE International Real-Time Systems Symposium RTSS 2007*. 49–60.

SCHOEBERL, M. 2004. A Time Predictable Instruction Cache for a Java Processor. *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004) 3292*, 371–382.

Tasking 2005. *Tricore v2.2 C Compiler, Assembler, Linker Reference Manual*. Tasking.

THIELE, L. AND WILHELM, R. 2004. Design for time-predictability. In *Design of Systems with Predictable Behaviour*.

UNGERER, T., CAZORLA, F., SAINRAT, P., BERNAT, G., PETROV, Z., ROCHANGE, C., QUINONES, E., GERDES, M., PAOLIERI, M., WOLF, J., CASSE, H., UHRIG, S., GULIASHVILI, I., HOUSTON, M., KLUGE, F., METZLAFF, S., AND MISCHE, J. 2010. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro 30*, 66–75.

WANG, D., GANESH, B., TUAYCHAROEN, N., BAYNES, K., JALEEL, A., AND JACOB, B. 2005. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News 33*, pp. 100–107.

WOLF, W. 2007. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann, San Francisco, CA, USA.